Spring 5-2-2021

Modernizing Legacy Business Practices and Maintaining Backwards Compatibility When Replacing Legacy Software

Thomas Hillebrandt

MODERNIZING LEGACY BUSINESS PRACTICES AND MAINTAINING BACKWARDS COMPATIBILITY WHEN REPLACING LEGACY SOFTWARE

An Undergraduate Honors Thesis
Submitted in Partial fulfillment of
University Honors Program Requirements
University of Nebraska-Lincoln

by
Thomas Hillebrandt, BS
Computer Science
College of Arts and Sciences

05/02/2021

Faculty Mentor:
Christopher Bohn, Ph.D., Computer Engineering

Abstract

As technology advances and hardware as well as user expectations becomes more advanced, software systems must change alongside or go obsolete. When software is no longer developed, decisions must be made regarding its future. Through various methods, legacy software may continue to see usage far past its obsolescence, however legacy software will sooner or later face replacement by new applications, built for state-of-the-art machines, to comply with modern requirements.

When writing new software to replace older programs, the added challenge for developers is to help the client also modernize their workflow. When a program has been in long time use - sometimes for decades - there may be a tendency to lean towards a “we’ve always done it this way” attitude. Replacing legacy software should come with some level of replacing legacy practices, while maintaining necessary backwards compatibility with any other legacy software the old program would interact with. It is important to ensure that replacement software doesn’t just become the old program in a shinier box, but rather can be an elegant product, for a more civilized age.

Keywords: Backwards Compatibility, Computer Science, Legacy Code, Legacy Software, Legacy Workflow, Requirements Elicitation, Software Engineering,

Acknowledgements

I want to acknowledge the invaluable help of Dr. Christopher Bohn and Dr. Suzette Person in finding my direction on this thesis. I furthermore must acknowledge the constant support from my wife, Olga, as well as her offering of one personal experience regarding an example of patching legacy software.

Table of Contents

MODERNIZING LEGACY BUSINESS PRACTICES AND MAINTAINING BACKWARDS COMPATIBILITY WHEN REPLACING LEGACY SOFTWARE

1 Introduction

As technology advances and hardware becomes faster, more advanced, and more complex, software systems must change alongside or go obsolete. And the software must keep up, not just with advances in hardware, but also with the expectations and needs of its users. Many programs eventually see the end of their natural life cycle, when further development no longer takes place, either due to technical issues, lack of documentation, unavailability of knowledgeable developers, or simply due to it being cost prohibitive. Such legacy software is dealt with in a number of different ways, ranging from simply pretending the problem isn't there, through adding a patchwork of software on top to keep the legacy code running, to replacing it with new software.

When a new application is being developed to replace a legacy program, the developer should make a concerted effort to help the client also update their legacy workflow, developing a product that fulfils the client's needs, brings their business practice as well as their code into the present, while maintaining compatibility with any other software the client relies on.

1.1 Moore's Law and Software

In 1965, Gordon E. Moore, then at Fairchild Semiconductors, predicted that the number of transistors on a single integrated circuit would double every year [1], a prediction later to become known as Moore's Law, and one he would revise a decade later to be a doubling of transistors every 24 months [2]. Subsequent revisions of Moore's prediction notwithstanding,

it must be acknowledged that, by and large, he was onto something: He had predicted that by 1975, microchips would have 65,000 transistors; by 1975, a new series of memory chips contained 65,536 transistors [3]. By the release of Intel's Third Generation Core processors in 2012, 1.48 billion transistors were crammed onto a single chip [4]. And as integrated circuits and microchips have been increasing in complexity, so has the software built to interact with them.

Not only has the software needed to keep up with the more advanced hardware available, but user demands for what could and should be possible with software have increased. Where, at the dawn of personal computing, computer use was fairly exclusively relegated to industry, science, and eager hobbyists, computers — and their software — have seen increasingly wide use as time has progressed. From word processing, home banking, art and music, through games and portable entertainment devices, to social media, e-commerce, the internet, and embedded systems controlling everything from home lighting and thermometers, to cars and security systems, computers, and the software to run them, have found a place in virtually all corners of society [5]. And so, after decades of advances, multitudes of programs, created at various stages of hardware complexity, still exist. Some have been continuously developed and updated to keep up with technology, while others have fallen by the wayside. And when a piece of software is developed no longer and becomes legacy, decisions must be made regarding the operations it performed, and about these operations should be carried out in future.

1.2 The Legacy Software Problem

Legacy software has been defined in various more or less different ways. For the purposes of this paper, legacy software shall be defined simply as software that is still in use, but is no longer actively being developed, though the software may still see adaptive maintenance, such

as varying levels of wrapping, patching, etc., (see section 2.1), and in rare cases even corrective maintenance (fixing catastrophic errors).

Often software is the result of multiple iterations of updates and adaptations, layer atop layer of functionality, leading to a sometimes bloated, but usually well-tailored program suite. As time passes active development may cease, and some different form of maintenance will be required to keep the software functional.

So why keep the software in use if it's bloated, no longer being developed, and difficult to maintain? Typically, legacy software is kept in commission due to its fulfilling a central business function. Interviews and surveys have found that, in many cases, legacy systems are kept around because they form the core of a business, are regarded as stable, well tested, and reliable, as well as the habits of users, unwillingness to change, and concerns about policy [6][7].

So then, why not just keep the software in use, if it is stable and well tested, if it performs its business functions? Inevitably, a point will be reached where the maintenance required to keep the software working is no longer practical, or even possible. Turnaround time for updates to patches or wrappers may be too slow; the legacy hardware required to run the software may be impractical or impossible to keep working; original developers may not be feasibly available to write patches for the original program; documentation may be scarce or absent; source code might no longer exist, or it may be written in a language no longer in wide use, making finding someone proficient enough to undertake the maintenance an insurmountable task. After all, thousands of programming languages have been invented since the dawn of computing. Just in the US Department of Defense, over one thousand languages were in use by the 1980's [8], and very few of them see much use, if any at all, today.

While various methods have been — and are being — used to keep old software in action, sooner or later stakeholders are going to want to explore new avenues to keep their business running smoothly. Legacy software — eventually — will need to be phased out. In some cases, adequate replacement software exists in the form of Commercial-Off-The-Shelf (COTS) products. But even if such a COTS application can perform most of the functions needed from the legacy program, it may not be easily compatible with other pieces of software used alongside that being replaced. File and API compatibility needs to also be addressed. It may make the most sense to commission a new piece of software to replace the program being retired. And in such a case, exactly what this new software should do, and how, is no less important than any other new application suite, but may prove more difficult to determine.

2 Dealing with Legacy Software

When software approaches legacy, the first step the user contemplates isn't necessarily to commission a new piece of software to replace it. Due to costs, time, habit, or some other perceived obstacle, the users may choose to keep the software in use, despite lack of continued development and support.

2.1 Keeping Legacy Software in Use

In the real world, various solutions can be, and have been, employed to allow the continued use of otherwise obsolete software:

Ignore It: If the software still runs on current systems, or if the users are able and willing to keep a less-than-current system around for the purpose, and if the software can perform the needed tasks this way, an obvious solution may be to simply ignore the problem. "If it ain't broke, don't fix it." However, this is unlikely to remain a valid solution for very long. It's likely

the evolution of surrounding systems eventually will cause problems with the legacy software, in terms of compatibility or other interface issues, or — in the case of keeping old systems around to run the old software — keeping that old system running may prove to be a greater challenge than expected. From personal experience, an engineering firm in the early-to-mid aughts used MS-DOS PC's to manage their photocopier log-ins. Every user had to log in to make photocopies, such that any billing would go to the correct project. This worked well, until they ran out of replacement DIN-plug keyboards for the old 286 machines.

Patch It: If the program runs, but input and/or output datatypes are no longer in wide use, another short-to-medium-term solution may be to add intermediate software between the legacy program and the system it runs on. Conversion tools can allow the software's output to be used in modern applications and vice versa. Of course, this solution not only adds extra, intermediate steps, which slows down the workflow, it also introduces extra steps where mistakes might happen. Another example from personal experience, a data journalism institute in the mid-2010's offered a service where they would clean government data, so it could be used in publication. The institute relied on a plethora of scripts written in FoxPro to perform the cleaning, but because FoxPro had been phased out about a decade earlier, the cleaned data would have to then first be converted to CSV, before it could be loaded into a modern application for use.

Wrap It: If the software no longer runs on modern systems, and keeping an old machine around specifically for the program is impractical (which it seems would always be the case), another option would be to wrap the entire legacy application in one or several emulation layers, allowing software that is technically incompatible with modern operating and networking systems to nonetheless run. A relatively simple and common example of this is running MS-DOS applications in DOSBox on modern Windows (or even Linux or MacOS [9])

systems. This can significantly extend the longevity of outdated software, especially if combined with the previous option. But ultimately, stakeholders are likely to find themselves wishing for something more streamlined, something less likely to cause compatibility issues, and something with a lower risk for mistakes.

Eventually it will make the most sense to retire the legacy code and replace it with something new, either in the form of a COTS application, or with a new program written expressly to replace the legacy program.

2.2 Building on a Legacy Foundation

Depending on the state and structure of the legacy software in question, it may fall under consideration to reuse elements of the existing codebase. This possibly could offer some advantages. In some cases, it may save time and effort, and by extension cost, since it allows developers to do little to no work on certain aspects that won't need to change significantly. Since the reused code has already been in production, and undergone whatever bug fixes may have been necessary through time, it may reduce the potential for error in the renewed application. And, not insignificantly, it could conceivably feel reassuring to the client, that some important aspects of the program will be familiar and work as expected. Conversely, such software recycling could end up being as big of a challenge as simply starting over. Depending on the age of the program, and on what language it was written in, it may be an entirely monolithic entity which is hard to split into usable segments. Ample re- and reverse engineering may well be required to produce necessary documentation, which in turn could end up being as complex as the code it's trying to document [10]. But, if the codebase is sufficiently modular, even if the final goal is to have a completely new program suite, replacing "bits and pieces" along the way may be the safer approach, especially if the original system is very large.

But building on old code doesn't come without its own risks and potential pitfalls. The developer must be aware of every aspect of the original program, so as to avoid the new code conflicting with the old. New routines written to respond to certain signals from the legacy code may fail to respond, or respond correctly, to unexpected, undocumented — but no less important — signals. And, crucially, if developers aren't careful, old code may respond unexpectedly to signals from the new. A financial services firm in 2012 had sought to replace part of their software with new code that would increase the speed of bulk trading. The developers had — carelessly — reused an old signal flag that they had thought to be unused. It wasn't. The flag was part of a test algorithm which, while officially no longer being used, was still present in the legacy code. When they rolled out the new code, implementation failed on one of their servers — causing the old test algorithm to be activated by the reused flag. It took them less than half an hour to deactivate the malfunctioning code. By then it had cost them nearly half a billion US dollars [11]. Several poorly planned aspects added up to this massive error, and the lack of any form of version control and automatic deployment system certainly did not help. Had they followed best practices, and not left "dead" code in, properly documented the code base they were working on top of, and not reused old activation flags under the mistaken assumption that their former use was phased out, they would not have ended up in such an unfortunate situation.

Reusing elements of legacy can certainly be beneficial, but must be done so very carefully. In cases where the new program needs to be able to communicate with older programs, either directly, or through the import and export of files, reusing old code may be especially beneficial. More about this is section 3.2.

3 Replacing Outdated / Obsolete Software

When a developer is commissioned to produce a new piece of software to replace the functions of a legacy program, the initial approach may not seem altogether far removed from software engineering standard practice. Meetings are held with the stakeholder, and a plan is made for what shape the project should take. As far as the stakeholder is concerned, however, it may seem like half the work is already done; there is a legacy application "template" to work from. The developer must now take care to help the client design an application that modernizes their business practices, along with the software.

3.1 Requirements Elicitation

The challenge facing the developer during requirements elicitation, especially in the early stages, is, as Watts Humphrey observed, that "users do not know what they want a software system to do until they see it working" [12]. A client may have grand and detailed ideas about what exactly they think they want the software to do. Binders may be compiled with wishes and requirements, and handed to the development team with a "get coding" attitude, only for them to later find they were asking for things that they wouldn't need, or wouldn't need down the line. When the FBI in the early aughts found they needed a centralized case management system, they did just that; they went with the waterfall approach (twice), compiling vast amounts of requirements up front, and spending hundreds of millions of US dollars, before eventually writing off the project and starting over with an agile approach [13]. Alternatively, a client may have only vague, undetailed ideas about what exactly they need the software to do, and it then befalls the developer to poke and prod to get the client to be more specific — a task that is ideally helped along through the use of wireframes, prototypes, and iterative demo releases.

But, as tricky it can be to tease out detailed requirements from a client, trickier still, when designing a piece of software to replace one that has been in use for perhaps decades, can it be to adjust the expectations of a client who is fixated on how the legacy software is operated. This client already knows what the program should do, and how it should work, and may be perfectly content to throw the developers in halfway down the waterfall model, and expect them to essentially just start coding.

3.1.1 Considering Workflow

During any requirements elicitation, it is of course important that the workflow of the end-user is considered first and foremost. It is after all they who will be working with the software. This activity can, to some extent, be performed through a simple interview; ask the end-user(s) to define and describe how they want to use the software. In the case of replacement software for a legacy application, the end-users can be asked to define and describe how they use the current application. Nobody knows the ins and outs of how to "do the thing" better than those who are doing the thing. Observing users in action is another way of learning what the requirements are. And, again, in the case of legacy software, observing seasoned users can certainly provide a very detailed picture of what the workflow looks like. Even so, in the case of both interviews and observation, there is the possibility that the users will miss details, simply because they are too natural a part of the workflow — they have become second nature, akin to scratching an itch. It has been suggested to perform more unobtrusive observation of users, where the users aren't immediately aware they are being observed, to achieve a greater context-based image of their workflow [14]. This would indeed eliminate the omission-by-habit that might otherwise occur when the user is "performing" for the developers.

3.1.2 Legacy Software, Legacy Workflow

However, regardless of which techniques are or aren't employed in an effort to elicit the requirements for a replacement piece of software, one danger remains: Carrying over antiquated workflow. Where there may be a great desire to update the software in one aspect, there may be great resistance to update it in those that affect the "traditional" business practices [7]. It is incumbent upon the developer to be aware of such instances, and to be able to argue in favor of modernizing the habits of the end-user where appropriate. It may be a tight-rope walk at times, because of course the developer wishes to please the client. Ultimately, the client may end up more pleased when the new system is in line with the state-of-the-art, even at the cost of relearning how certain operations are performed. To put it on a point, there is little benefit in modernizing a system, if the client demands: "But how do we save to floppy disk?"

To consider a somewhat less absurd example, also from personal experience: A client commissioned a new data entry application to replace a legacy program. The old MS-DOS application used flat files, stored locally, and its user interface used arrow keys to navigate between input fields, rather than the modern standard of TAB/Shift-TAB. The clients expected similar behavior and functionality from the new application — their main concern was the ability to maximize the screen size, since the DOSbox wrapper wouldn't let them do that to the old program. Now, adding arrow key navigation to the new application shouldn't be a big problem — why upset the status quo? But, to reference the Principle of Least Astonishment [15], users expect systems to act in specific ways. And while "veteran" users of the legacy system may not be remarkably astonished by input field navigation done by arrow keys, new hires are very likely, given the ubiquity of the TAB/Shift-TAB method, to wonder why that doesn't work as expected. Of course, a relatively simple UI feature like that could simply

feature both functionalities, and neither seasoned nor green users would spare it a second thought. That said, it may well behoove a requirements-eliciting developer to underline to the client the added time — and cost — incurred by "jerryrigging" a likely standard UI library, just to re-introduce old-fashioned behavior. So how about file storage? The client expected flat files on their local drive, out of habit, but also out of a "fear of the web." Dealing with sensitive, IRB and HIPAA compliant data, they adamantly insisted that storing things "in the cloud" was far too risky. They felt safer "stuffing their cash in the old mattress." In this case, writing software that used local flat files for data would not require the developers to add obsolete layers to a standard library. Nonetheless, writing a modern application not using the modern technologies available, would not ultimately serve in the client's best interest, despite their insistence. The developer should be prepared to advocate for modernizing the client's attitudes and workflow practices, as well as the software, and assuage any concerns the client may have about safety and privacy in the cloud.

3.1.3 Reaching an Understanding

While it can be said to be the developer's job to know the state-of-the-art, and the client's job to explain the state-of-the-practice, in order for detailed requirements to be drawn, it is obviously important that the client ultimately feels listened to. The developer might decide "they know best" and simply overrule the requests of the client [16]. And while this paper is arguing that it is important that developers help lift clients out of "legacy habits", it must be done in a fashion that leaves the client feeling respected and — crucially — heard. It is as easy for the novice developer to impose their opinions on a project as it is for the client to insist on doing things the way it's always been done. Modernize the workflow, but do so with the blessing and understanding of the stakeholders, which demands the developer understanding

the stakeholders. In the end, the client will be using the new software, and to do so comfortably should feel the product is what they asked for [17].

3.2 Backwards Compatibility

It may seem as a contradiction, having discussed the merits of modernizing the clients' workflow as part of requirements elicitation, to then talk about the need for replacement software to be backwards compatible with the software it replaces. There are, however, some aspects that should be considered.

3.2.1 Retaining Old Data

In most, if not all, cases, replacing a piece of legacy software with a new application does not mean that all data from the old software can just be tossed out. There may be customer records that need to be carried over, invoice records that need to be accessible, or production files that are still in use. Depending on the type of data, it is likely to be of vital importance to the stakeholder that it can be transferred to a new system. The balance, here, becomes between built-in backwards compatibility; where old files can effortlessly be opened by the new software — or conversion; where a separate conversion tool is produced or acquired, to once-and-for-all convert all old data into the new program's format and be done with it. In cases where the old data is composed of things like records, conversion is likely to be the cleanest way to go. This one-time operation gets the client ready for the new application, without leaving henceforth unneeded code floating around in the otherwise shiny new application. When dealing with other types of data files used by the program for various purposes, it may make more sense to build in backwards compatibility. This could be especially true if future influx of files in the legacy format is likely to occur, such as from other programs part of the business workflow.

3.2.2 “Sideways” Compatibility

But backwards compatibility isn’t just a question of being able to read old files. It can be equally important that the new software can output data in a way that other, older programs require to handle it. It’s probably not likely to be the case that files produced by the replacement software will need to be read into the software that was replaced, but a secondary program suite that uses the output from the legacy program will need to accept output from the replacement software. While this could also be accomplished with stand-alone conversion tools, if the secondary application is going to continue being used for the foreseeable future, adding extra conversion steps isn’t likely to please the client, not to mention the potential for mistakes that introduces, its being quite close to the Patch It solution discussed in section 2.1. Building in the ability to produce the desired files — even if the format is somewhat outdated — will make for a more streamlined user experience.

In some cases, a tempting solution may be to achieve this “sideways” compatibility by using a general standard file format. Tabular data, for example, can be read by many varied applications in CSV (Comma Separated Value) format. Depending on the target application however, this could result in data loss, and would almost certainly still add steps for the end user.

As mentioned in section 2.2, where possible, the solution to such sideways compatibility may be the reuse of elements of the original codebase. Some legacy data formats have been around for so long that, in order to keep them consistently backwards compatible but simultaneously operational with modern data, their structure is anything but straightforward. Having a ready-made piece of legacy code that can convert simple data into such a format may be highly preferential to trying to rebuild the data structure from scratch.

3.2.2 Maintaining API Compatibility

Of course, compatibility is not all about files. The software may interact directly with other programs, or with web services, through an API. Whether such interaction is incoming or outgoing, the new application must effectively perform and/or receive such API calls. This may even present another opportunity for modernization. Libraries evolve and are developed upon, and certain features may end up deprecated [18], and where the legacy application may be calling a deprecated feature, the new application can update this.

In the inverse case, where other applications or services are calling the legacy software, it is no less important that — even if some feature or functionality is now regarded as deprecated — that the new application responds in a useful manner to such a call. In the same way that "sideways" compatibility requires the new software to be able to save in a perhaps arcane file format, so must it be able to handle a call to an arcane feature.

4 Conclusion

Legacy software is never going away. Today's brand new tool is tomorrow's legacy workhorse. As we progress, however, and developers write more modular code, this code will be much easier to handle when it becomes legacy and needs to be updated or replaced.

When writing new software to replace legacy code, it is naturally important to listen to the client, the experienced users of the program that is being replaced, to learn every detail about what the program needs to do, and how it needs to do it. But to avoid the new application becoming obsolete before it's even done compiling, it sits upon the developer to guide the users past legacy habits that could no less benefit from modernization, as can the code being replaced.

References

[1] G. E. Moore, “Cramming More Components Onto Integrated Circuits,” *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82-85, Jan. 1998, doi: 10.1109/JPROC.1998.658762, https://ieeexplore.ieee.org/document/658762 [Accessed Dec. 22, 2020].

[2] I. Tuomi, “The Lives and Death of Moore’s Law,” *First Monday*, vol. 7, no. 11, Nov. 2002, https://journals.uic.edu/ojs/index.php/fm/article/view/1000/921 [Accessed Dec. 23, 2020].

[3] Intel Newsroom, “Intel at 50: Moore’s Law,” *Intel*, 2018. [Online]. Available: https://newsroom.intel.com/news/intel-at-50-moores-law/. [Accessed Feb. 3, 2021].

[4] Intel, “Moore’s Law: Fun Facts,” *Intel*, 0312/TM/LAI/PDF 326942-001US, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/history/history-moores-law-fun-facts-factsheet.html. [Accessed Feb. 3, 2021].

[5] M. R. Swaine, W. M. Pottenger, P. A. Freiberger, and D. Hemmendinger, “Computer: History of Computing,” *Encyclopedia Britannica*, n.d. [Online] Available: https://www.britannica.com/technology/computer/History-of-computing. [Accessed Feb. 5, 2021].

[6] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage, “How do professionals perceive legacy systems and software modernization?” *Proceedings of the 36th International Conference on Software Engineering*, pp. 36–47, 2014. doi: 10.1145/2568225.2568318, https://dl.acm.org/doi/10.1145/2568225.2568318 [Accessed Dec. 18, 2020].

[7] A. Alexandrova, L. Rapanotti, and I. Horrocks, “The Legacy Problem in Government Agencies: An Exploratory Study,” *Proceedings of the 16th Annual International Conference on Digital Government Research (dg.o ‘15,)* New York, New York, pp. 150-159, May 2015, doi: 10.1145/2757401.2757406, https://dl.acm.org/doi/10.1145/2757401.2757406. [Accessed Feb. 15, 2021].

[8] D. A. Cook and E. Bingue, “Ada - A Failure That Never Happened!” *CrossTalk - The Journal of Defense Software Engineering*, vol. 30, no. 1, pp. 4-7, Feb. 2017, https://community.apan.org/wg/crosstalk/m/documents/261538 [Accessed Jan. 14, 2021].

[9] DOSBox, *DOSBox*. [Online]. The DOSBox Team, 2002-2021. https://www.dosbox.com/information.php.

[10] H. M. Sneed, “Recycling Software Components Extracted from Legacy Programs,” *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE ‘01,)* New York, New York, pp. 43-51, 2001, doi: 10.1145/602461.602469, https://dl.acm.org/doi/10.1145/602461.602469. [Accessed Feb. 13, 2021].

[11] H. Dolfing, “Case Study 4: The $440 Million Software Error at Knight Capital,” *henricodolfing.com*, Jun. 5, 2019. [Online]. Available: https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html. [Accessed Feb. 12, 2021].

[12] C. Jones, “A Retrospective View of the Laws of Software Engineering,” *CrossTalk - The Journal of Defense Software Engineering*, vol. 30, no. 1, pp. 17-23, Feb. 2017, https://community.apan.org/wg/crosstalk/m/documents/261538 [Accessed Jan. 14, 2021].

[13] B. Wernham, “FBI Sentinel Programme Saved by Agile?” *brianwernham.wordpress.com*, May 31, 2012. [Online]. Available: https://brianwernham.wordpress.com/2012/05/31/fbi-sentinel-programme-saved-by-agile-2/. [Accessed Feb. 12, 2021].

[14] A. Knauss, “On the usage of context for requirements elicitation: End-user involvement in IT ecosystems,” *2012 20th IEEE International Requirements Engineering Conference (RE)*, Chicago, IL, pp. 345-348, 2012, doi: 10.1109/RE.2012.6345835, https://ieeexplore.ieee.org/document/6345835 [Accessed Jan. 20, 2021].

[15] P. Seebach, “The Cranky User: The Principle of Least Astonishment,” *IBM developerWorks*, Aug. 2001, https://www.ibm.com/developerworks/web/library/us-cranky10/. [Accessed Feb. 11, 2021].

[16] L.-A. M. Kastman Breuch, “The Overruled Dust Mite: Preparing Technical Communication Students to Interact with Clients,” *Technical Communication Quarterly*, vol. 10, no. 2, pp. 193-210, 2001, doi: 10.1207/s15427625tcq1002_5, https://www.tandfonline.com/doi/abs/10.1207/s15427625tcq1002_5 [Accessed Jan. 24, 2021].

[17] P. Kimmerly, “Process is Easy, Change is Hard,” *CrossTalk - The Journal of Defense Software Engineering*, vol. 30, no. 2, pp. 22-23, Mar. 2017, https://community.apan.org/wg/crosstalk/m/documents/261537 [Accessed Jan. 15, 2021].

[18] A. A. Sawant, G. Huang, G. Vilen, S. Stojkovski, and A. Bacchelli, "Why are Features Deprecated? An Investigation Into the Motivation Behind Deprecation," *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid, 2018, pp. 13-24, doi: 10.1109/ICSME.2018.00011. https://ieeexplore.ieee.org/document/8529833. [Accessed Feb. 14, 2021].